Towards a flexible LOPS implementation

An example of XPRTS programming

This work has been funded by ESPRIT

Christoph Kreitz
Institut für Informatik
Forschungsgruppe Intellektik
Technische Universität München

Table of contents:

Introduction: **Scope of the paper**

Ground level: **Basic operations**

0.1 Extension of XPRTS build-in primitives
0.2 Operations on terms for first-order-logic
0.3 Operations on LOPS-formulae and literals
0.4 Knowledge Base Interface
0.5 Primitive transformations for execution within LOPS strategies
0.6 Theorem Prover Interface - Extended Matching

Level 1: **Elementary transformations**

1.1 Equivalence Transformation Procedure
1.2 Normalization procedures
1.3 Guessing
1.4 Syntactical and logical reduction (Outlook)
1.5 Transformation into recursive form
1.6 Handling unevaluable predicates (Outlook)
1.7 Reducing the number of output-variables (Outlook)

Outlook: **Level 2 and higher levels**

Appendix: **Some Demo Syntheses**

A.1 Maximum example
A.2 Selection sort

References

Introduction: Scope of the paper

This paper describes an experimental approach to an implementation of a Program Synthesis System based on the system LOPS (see ***[Bibel 1980, Bibel/Hoernig 1984]***) using the XPRTS Program Synthesizer Generator. As in the previous LOPS implementation the key idea is to apply equivalence transformations to a formula until an executable (and hopefully efficient) program is reached. Contrary to the previous system, however, the current implementation focuses on being flexible and easy to extend by using domain knowledge much more explicitly than before. In addition its strategies are organized in a hierarchical structure of abstraction levels. Lower level operations will be operations which transform formulae according to parameters given by the user while strategies from the higher levels will contain more and more heuristics how to select these parameters. In a sense the hierarchy is open-ended. Experiences from example syntheses may lead to insights on new methods which eventually will result in the implementation of strategies of a higher level. The only exception to that is the general strategy of LOPS (figure ***GN 1/87***) which will be the top-level of the system.

The experimental implementation of strategies is done within the X-language described in the "user's manual for XPRTS" ***[XPRTS]*** which together with some useful extensions is considered to be the ground level of the Program Synthesis system. For efficiency reasons, however, strategies once established should later be coded in C. In the paper we describe the current state of the implementation and give an outlook for work to be done. Right now level 1 of our prototype is being implemented. Higher levels are in preparation.

Ground level: Basic operations

At the ground level we collect some basic operations that are necessary for the analysis and synthesis of formulae to be used by the higher level LOPS-strategies. Low-level interfaces to a theorem prover and a knowledge base also belong to this level as well as the purely transformational parts of some standard strategies which for this reason have been separated from the other parts. Procedures at this level contain absolutely no heuristic components.

The set of these procedures defined is by no means complete yet.

0.1 Extension of XPRTS build-in primitives

Since lists are the main data structure of XPRTS some list operations might be helpful for writing strategies in a readable fashion.

All procedures given here are purely functional, i.e. except for possible failure they have no side effects. Results, as usual, appear in the accumulator $$ which allows to use them immediately as input for other procedures. If multiple results are needed they are returned as lists of results "[r1,r2,..,rn]". In general, boolean functions have no result. "false" is indicated by failure of the function "true" by successful execution. According to the standard XPRTS philosophy these kinds of results are stored in the "done"-flag.

null (list)

Returns ***true***, if list is empty, ***false*** otherwise.

nonempty (list)

Returns ***true***, if list is not empty, ***false*** otherwise.

hd (list)

Returns the first element ("head") of the list.

tl (list)

Returns the tail of the list.

lg (list)

Returns the number of elements of the list. ***(now an XPRTS basic primitive)***

cons (element, list)

Inserts ***element*** at the beginning of the list and returns the result.

fst_of (list-of-lists)

Returns a list of the first elements of all the lists which are elements of ***list-of-lists***.

(list-extension of *hd****)***

reverse (list)

Returns the list in reverse order. ***(now an XPRTS basic primitive)***

append (l1, l2)

Appends the two lists ***l1*** and ***l2*** and returns the result. ***(now an XPRTS basic primitive)***

appendl (list-of-lists)

Appends all the lists which are elements of ***list-of-lists*** and returns the result.

(list-extension of *append****)***

0.1.2 Finding elements within lists

find (element, list)

If ***element*** is the n-th member of the list then the result is ***[n]***. The procedure fails if ***element*** does not appear in the list.

(now an XPRTS basic primitive for all kinds of items)

is_in (element, list)

Returns ***true***, if ***element*** appears in the list, ***false*** otherwise.

is_subset (l, L)

Returns ***true***, if every element of the list ***l*** also appears in ***L***, ***false*** otherwise. ***(now an XPRTS basic primitive)***

0.1.3 Selecting elements of a list

select (i, list)

Returns the i-th element of the list.

sel (indices, list)

Returns a list of elements of ***list*** as given by the list of indices.

(list-extension of *select****).***

getl (item, structure-description-list)

Returns a list of substructures of ***item*** as given by the list of structure-descriptions

(list-extension of the build-in primitive *get****).***

0.1.4 Elimination of elements

remove (i, list)

Removes the i-th element of ***list*** and returns the result.

del (element, list)

Removes ***element*** from ***list*** and returns the result, if ***element*** is a member of ***list***, fails otherwise.

diff (List, l)

Removes all the elements of the list ***l*** from ***List*** and returns the result.

(list-extension of *del* ***).***

elim (indices, list)

Removes elements of ***list*** given by ***indices*** and returns the result.

make_set (list)

Converts a list into a "set":

Returns a list of elements of ***list*** where multiple occurences are deleted.

0.1.5 Replacing elements

exchange (list, i, item)

Replaces the i-th element of the list by ***item*** and returns the result.

exchange_for (list, term, substterm)

Replaces every occurence of ***term*** in the list by ***substterm*** and returns the result.

0.2 Operations on terms for first-order-logic

In order to raise the programming level from "machine-oriented" tree-operations to rather logic-oriented ones we define some functions which later shall become the only way to manipulate terms for first-order logic. It is the intention that direct ways to access terms will be forbidden. There are three main categories of operations: constructors, destructors, and identifying predicates. Constructors take a list of items and construct the corresponding term from them. For instance, ***make_and([P(x),Q(y),P(z)])*** will result in ***(P(x)&Q(y)&P(z))***. Destructors perform the opposite. The components of a term will be returned in a list. In addition to that nested terms will be flattened out by destructors, e.g. ***destruct_and((P(x)& (Q(y)&P(z))))*** results in ***[P(x),Q(y),P(z)]***. Destructors will fail if applied to wrong term kind. Predicates identify the kind of term one has to deal with. (These procedures now make use of the new system primitives *change* and *typep*.)

0.2.1 Constructors

make_and (conjuncts)

On input ***[t1,t2,..,tn]*** returns the term ***(t1&t2&..&tn)***. Fails if conjuncts are an empty list.

make_or (disjuncts)

On input ***[t1,t2,..,tn]*** returns the term ***(t1|t2|..|tn)***. Fails if disjuncts are an empty list.

make_xor (disjuncts)

On input ***[t1,t2,..,tn]*** returns the term ***(t1 XOR t2 XOR..XOR tn)***. Fails if disjuncts are an empty list.

make_imp (p, q)

Returns ***(p -> q).***

make_if (p, q)

Returns ***(p <- q).***

make_equiv (p, q)

Returns ***(p <-> q).***

make_not (p)

Returns ***(~p)***

make_all (variable-list, term)

On input ***([x1,x2,..,xn],term)*** returns the term ***All x1 (All x2..(term)..)***.

make_ex (variable-list, term)

On input ***([x1,x2,..,xn],term)*** returns the term ***Exist x1 (Exist x2..(term)..)***.

0.2.2 Destructors

destruct_and (term)

On input ***(t1&t2&..&tn)*** (arbitrarily nested via parentheses) returns the list ***[t1,t2,..,tn]***. Returns ***[term]*** if top-junctor is not AND.

destruct_or (term)

On input ***(t1|t2|..|tn)*** (arbitrarily nested via parentheses) returns the list ***[t1,t2,..,tn]***. Returns ***[term]*** if top-junctor is not OR.

destruct_xor (term)

On input ***(t1 XOR t2 XOR..XOR tn)*** (arbitrarily nested via parentheses) returns the list ***[t1,t2,..,tn]***. Returns ***[term]*** if top-junctor is not XOR.

destruct_imp (term)

On input ***p -> q*** returns ***[p,q]***.

destruct_if (term)

On input ***p <- q*** returns ***[p,q]***.

destruct_equiv (term)

On input ***p <-> q*** returns ***[p,q]***.

destruct_not (term)

On input ***~p*** returns ***p***.

destruct_all (term)

On input ***All x1 (All x2..(term)..)*** returns the list ***[[x1,x2,..,xn],term]***.

destruct_ex (term)

On input ***Exist x1 (Exist x2..(term)..)*** returns the list ***[[x1,x2,..,xn],term]***.

Note that results of destructors are always lists (except for *destruct_not*), even if the corresponding constructor takes a pair as input.

0.2.3 Predicates

Predicates return ***true*** if a term ***t*** is of the requested kind, ***false*** otherwise. Since the names speak for themselves we just give the syntax.

is_and_term (term)	*is_or_term (term)*	*is_xor_term (term)*
is_imp_term (term)	*is_if_term (term)*	*is_equiv_term (term)*
is_not_term (term)	*is_all_term (term)*	*is_ex_term (term)*

0.2.4 Miscellaneous

flatten (term)

Flattens conjunctions and disjunctions, i.e. nested structures are being removed. ***(now an XPRTS basic primitive)***

0.3 Operations on LOPS-formulae and literals

Operations described here again shall raise the programming level to a rather logic oriented one. They are intended to deal with the formal problem descriptions within the LOPS program synthesis system, i.e. they will be the only way to analyze LOPS-formulae and literals or manipulate them for synthesis purposes.

0.3.1 LOPS-formulae

Before creating operations on LOPS-formulae we had to clarify in which way synthesis problems, i.e. problems which are to be transformed into an executable program, have to be represented in our system. There are several ways to do it:

I. ***PM <-> [All x1,..xn Exist y1,..ym IC(x1,..,xn) -> OC(x1,..xn,y1,..ym)]***

To solve a problem of this kind one must know that in fact it does have a solution. A program will be extracted from some constructive proof of the theorem PM. The disadvantage of this notation is that we cannot describe partial programs easily.

II. ***All x1,..xn,y1,..ym [PM(x1,..ym) <-> IC(x1,..xn) & OC(x1,..ym)]***

III. ***All x1,..xn,y1,..ym IC(x1,..xn) -> [PM(x1,..ym) <-> OC(x1,..ym)]***

We prefer version III which in the following will be referred to as LOPS-formula or briefly as **formula**. For simplicity quantifiers will be ommited. Free variables occuring in a formula are assumed to be all-quantified. Thus the general form of a formula in LOPS is:

Input-conditions -> [Problem <-> Output-conditions]

or, more generally stated: ***precondition -> [head <-> body]***

As already mentioned, the key idea behind LOPS is to apply equivalence transformations to a formula until an executable (and hopefully efficient) program is reached. Usually all the transformations operate on the body only which is assumed to be kept in disjunctive normal form. The precondition is necessary only for validity checks and the head often just represents the name of the program itself.

precond (formula)

Returns the precondition of the formula.

head (formula)

Returns the formula head.

body (formula)

Returns the formula body.

make_formula (precondition, head, body)

Returns ***((precondition -> (head <-> body))***.

replace_precond (formula, new-precondition)

Replaces the precondition of ***formula*** by ***new-precondition*** and returns the result.

replace_head (formula, new-head)

Replaces the head of ***formula*** by ***new-head*** and returns the result.

replace_body (formula, new-body)

Replaces the body of ***formula*** by ***new-body*** and returns the result.

0.3.2 Literals

make_literal (literal-name, arg-list)

Creates a literal with name ***literal-name*** and arguments given by ***arg-list*** and returns the result..

(now an XPRTS basic primitive)

0.4 Knowledge Base Interface

Currently the Knowledge Base of LOPS is simulated by the XPRTS-tape (see XPRTS manual) which also will be called "library". Information will be stored in it as list of the following format:

[<name of the object>, <type of the object>, <contents>]

The name should be a simple string identifying the information. For instance ***Lemma 1.3*** or ***max*** may be typical names. The type of objects is used to classify them. This may speed up search processes. Types may be ***lemma***, ***definition***, ***heuristic***, ***recursion scheme***, ***reduction lemma***, ***synthesis protocol***, etc. It is allowed to give the same name to different objects provided they are of different types. Obviously,

set(S) -> (max(S,m) <-> in(m,S) & leq(S,m))

are the contents of the definition ***max***.

The KB-operations yet defined are comparably primitive and need to be extended.

0.4.1 Elementary tape operations

load_tape

Loads tape from a standard file.

save_tape

Saves tape to standard file.

rewind

Rewinds the tape.

showlib

Shows names and types of all the objects in the library.

show_kb (type)

Displays the names of all the objects of the given type.

0.4.2 Storing and searching objects - general procedures

kb_add (type, name, contents)

Stores an object on tape. Currently no double-occurence check is done.

name_of (type, contents)

Given the type and the contents of an object *name_of* returns the name of the object if it exists in the Knowledge Base.

contents_of (type, name)

Given the type and the name of an object *contents_of* returns the contents of the object if it exists in the Knowledge Base.

search_topics (type, topics)

Searches for the next object of given type in which the named topics (i.e. predicates) are mentioned.

0.4.3 Storing and searching objects - type specific procedures

The type ***lemma*** stands for facts which are known to be true. Definitions will also belong to this type.

store_lemma (name, contents)

Stores a lemma in the KB.

find_lemma (contents)

Returns the name of a lemma with given contents.

lemma (name)

Returns contents of the named lemma if the name is valid.

search_lemma (topics)

Returns the next lemma in whose preconditions the named topics are mentioned.

The type ***rec_scheme*** describes knowledge about typical recursion schemes for certain domains. Recursion schemes are essential for most of the strategies used by LOPS (e.g. GUESS and GET-REC). See section 1.5 for more about recursion schemes

store_rec_scheme (name, contents)

Stores a rec_scheme in the KB.

find_rec_scheme (contents)

Returns the name of a rec_scheme with given contents.

rec_scheme (name)

Returns contents of the named rec_scheme if the name is valid.

search_rec_scheme (topics)

Returns the next rec_scheme in whose preconditions the named topics are mentioned.

The type ***synthesis*** is intended to store a protocol about performed program syntheses (which may be incomplete) in the Knowledge Base. A Synthesis Protocol is realized as a stack where the last entry will be on top. Every entry contains a description of the synthesis step (command) which was executed last and its result on basis of the previous entry. It is intended to use synthesis protocols for executing syntheses "by hand" as long as they cannot be handled automatically by the system and for analyzing performed syntheses in order to discover new strategies how to proceed. A synthesis starts with the problem description as contents and ***START*** as command description (see *start_synth*).

store_synth (name, contents)

Stores a synthesis in the KB.

find_synth (contents)

Returns the name of a synthesis with given contents.

synth (name)

Returns contents of the named synthesis provided the name is valid.

start_synth (name, contents)

Stores a problem to be synthesized in the KB by starting a synthesis protocol with the problem as contents.

problem_of_synth (name)

Returns the problem specification whose synthesis protocoll is stored under the given name

add_synth_step (name, command-description, result)

Adds another synthesis step to the named synthesis protocoll, if exists, by describing the comand executed and its result.

Procedures to show and edit synthesis protocolls will be defined as soon as the system is developed enough to perform syntheses with it with reasonable efforts.

0.5 Primitive transformations for execution within LOPS strategies

LOPS strategies as usually described often consist of a specific kind of equivalence transformation and a strategic part which selects the parameters to control these transformations. In order to separate the rather unintelligent part from the purely strategic one we define transformations with parameters as low level procedures.

0.5.1 Guessing

The GUESS-strategy modifies a certain part of a formula's body by introducing a new (all-quantified) variable ***z*** which corresponds to one of the output variables of the formula. A domain-condition DC for ***z*** and a tautology condition ***(P(z,yi)|˜P(z,yi))*** relating ***z*** and the output-variable ***yi*** (often ***P*** is equality) will be added. See chapter 1.3 for details.

t_guess_sd (formula, body-structure-description, dclist, tautology)

> Takes a formula and a structure-description for the part of the body on which guessing shall take place, a list of conjuncts for the domain-condition, and the tautology predicate and performs the transformation described above. The parameters will be selected by procedures in higher levels.

t_guess (formula, dclist, tautology)

> Similar to *t_guess_sd* but operating on the whole body (no structure-description needed).

0.5.2 Reduction

Reduction means elimination of superfluous parts of a formula's body in order to effectivize execution of syntheses. Higher level strategies will discover these superfluous parts and pass their structure-descriptions within a list to the executing transformation *t_reduce*.

t_reduce (formula, structure-description-list)"

0.6 Theorem Prover Interface - Extended Matching

As long as there is no connection to a "real" Theorem Prover verification of the satisfiability of theorems will be simulated by a subset-relation on conjuncts and an extended version of matching.

0.6.1 Extended matching

xmatch (item, template, sigma)

Tries to perform extended matching (see definition below) of ***item*** with ***template*** filling the substitution***sigma***. Returns the matching substitution if successful and fails otherwise returning the empty list.

Background:

Let I = ***(I1&I2&..&In)*** be an item and T = ***(T1&..&Tm)*** be a template
I *extendedly matches with* T using variables in vars iff there are indices ***j1, ..jm*** such that ***(T1&..&Tm)*** matches with ***(Ij1&..Ijn)*** in the conventional sense using variables in vars.

If I extendedly matches with T then satisfiability of T can easily be proven by I.

xlmatch (item, template, vars)

Performs an extended matching of two lemmata where the left and ride side are in clause form.

0.6.2 Verification of conditions

xverify (condition, item, vars)

Checks if .25condition is verifiable by ***item***. Currently this is considered to be the case if both the condition and the item are in clause form and all of the condition's conjuncts also appear in item or if item extendedly matches with condition using vars. *is_verifiable* is considered a boolean function which in addition returns a substitution list if necessary.

Later versions will call a Theorem Prover or directly consult the user. Will be extended to disjunctive normal form if there is a need for it (just check if one of the condition's clauses is satisfied by item).

0.6.3 Proving Lemmata by various means

The following procedures are currently being developed.

provable (openlemma, openpredname)

Tries to prove the open lemma by first asking the Knowledge Base, then a Theorem Prover, and finally the user. Open in this context means that a predicate occurring (which is given by openpredname) may be instantiated. If the open lemma is provable the result will be an instantiated version of it.

proven_by_kb (lemma, sigma, openpred)

Tries to find a lemma in the Knowledge Base that extendedly matches lemma using substitution sigma and also instantiates the open predicate. Extended matching for lemmata will be formally defined soon.

proven_by_tp (lemma, sigma)

Currently no Theorem Prover is available. The procedure simply fails.

proven_by_user (lemma, sigma)

Currently no interactive user input is possible. The procedure simply fails.

Level 1: Elementary transformations

At level 1 LOPS should be able to perform elementary formula transformation on some higher logical level. However, they still do not contain heuristic components. They are intended to be elementary strategic transformations whose parameters are to be controlled by heuristics of some higher level. We try to split the strategies occuring in the previous ("old") LOPS system into these major parts. The advantage will be that we first can gather some experience with formula transformation/ program synthesis "by hand" before we actually go on programming the heuristic part of the system. For better control every transformation can also operate on a certain part of a formula only.

1.1 Equivalence Transformation Procedure

transform_by (Lemma, structure-description, direction, formula)

Applies a lemma in formula-form to a certain part of the formula's body given by the structure-description according to the named direction and returns the result.

In case of a positive direction (***L*** - from left to right) application of a lemma has the following meaning:

In any clause of the given part of the formula's body where one of the clauses of the lemma's head can be instantiated it will be replaced by the instantiated version of the lemma's body provided the precondition holds. In case of a negative direction (***R***) head and tail are exchanged.

Formally spoken: Let the lemma head be $(H_{1,1}\&..\&H_{1,n1})\ |..|\ (H_{k,1}\&..\&H_{k,nk})$

and the selected part of the formula body be $(F_{1,1}\&..\&F_{1,m1})\ |..|\ (F_{j,1}\&..\&F_{j,mj})$.

If $(F_{l,1}\ \&\ ..\ \&\ F_{l,ml})$ extendedly matches some $(H_{i,1}\ \&\ ..\ \&\ H_{i,ni})$ and the instantiated lemma precondition can be verified within $(F_{l,1}\ \&\ ..\ \&\ F_{l,ml})$ and the formula's precondition then the matching conjuncts $F_{l,a1}..F_{l,ani}$ will be replaced by the instantiated body of the lemma. The procedure fails if no application of the lemma is possible.

For convenience some particular applications of transform_by have been defined as separate functions. Folding a lemma stands for application in negative direction, unfolding for positive direction.

fold_lemma (Lemma, formula, structure-description)

Apply *transform_by* in negative direction

unfold_lemma (Lemma, formula, structure-description)

Apply *transform_by* in positive direction

fold_sd (name, formula, structure-description)

Apply *transform_by* in negative direction using a lemma of the Knowledge Base with the given name

unfold_sd (name, formula, structure-description)

Apply *transform_by* in positive direction using a lemma of the Knowledge Base with the given name

Apply *fold_sd* using the whole body of ***formula***

unfold1 (name, formula)

Apply *unfold_sd* using the whole body of ***formula***

lemmata (lemmalist, formula)

Consecutively applies *transform_by* to formula using lemmata, structure-descriptions and directions as given by lemmalist.

1.2 Normalization procedures

Normalization procedures simplify terms by transforming them into disjunctive normal form or some intermediate form. Often it is enough to use the simpler procedures since the input term already is close to disjunctive normal form. The functions described here are rather ad hoc definitions and need improvement if the size of terms calls for effective transformations.

make_norm (term, structure-description)

The basic normalization transformation in parametrized form. Given a term and a structure-description of a subterm which is a disjunction *make_norm* returns a disjunction of terms which consist of the original term with the subterm replaced by either of its disjuncts.

For instance *make_norm* ***((A&B & (C1|C2|C3) &D),[3])*** will result in

((A&B&C1&D) | (A&B&C2&D) | (A&B&C3&D))

first_disjunct (conjunction)

Searches for the first conjunct which is a disjunction and returns its index. Fails if there is none.

make_normal (conjunction)

Creates a disjunctive normalization of the conjunction up to depth 2. The procedure *make_norm* (with structure-description chosen by *first_disjunct*) is repeatedly applied as long as possible.

For instance *make_normal* ***((A& (B1|B2) & (C1|C2|C3)))*** will result in

((A&B1&C1) | (A&B1&C2) | (A&B1&C3) | (A&B2&C1) | (A&B2&C2) | (A&B2&C3))

make_dnf (term)

Obviously, this procedure transforms a term into full disjunctive normal form.

normalize (formula)

dnf (formula)

These two procedures were written to ease the application of normalization procedures to formula transformations. They operate on the body of the formula only. *normalize* applies *make_normal* to the formula's body and *dnf* does the same with *make_dnf.*

1.3 Guessing

Guessing is one of the most important basic strategies of LOPS. Given a problem in which an output-variable ***yi*** cannot be determined immediately from the input variables guessing provides a reasonable way to find the output by trial and error methods. In order to restrict such attempts somehow it is required that the **guess-variable *g*** satisfies some domain-conditions which are a proper subset of the conditions on ***yi***. If guessing succeeds then ***yi*** can immediately be derived from ***g*** which often means ***yi=g***. In the other case we hope to achieve some recursive description of the problem by further strategies.

Thus guessing creates a guess-variable ***g*** and tries to find an appropriate domain-condition for it which is chosen from the conditions on the output-variable ***yi***. In addition to that a tautological relation between ***yi*** and ***g*** is introduced to express success and failure case. In nearly all applications this relation is ***yi=g|˜yi=g*** but there may be other choices. Since the transformation itself is obvious the most crucial part of guessing is to find the domain-condition and, in some cases, a more appropriate tautology predicate than equality.

Formally the transformation is derived as follows:

OC(x1,..ym) <-> ***OC(x1,..ym) & (DC(..g..) | ˜DC(..g..)) & (yi=g |˜yi=g)***

<-> ***OC(x1,..ym) & DC(..g..) & (yi=g |˜yi=g)***

| OC(x1,..ym) & ˜DC(..g..) & yi=g ***(1)***

| OC(x1,..ym) & ˜DC(..g..) & ˜yi=g ***(2)***

<- ***OC(x1,..ym) & DC(..g..) & (yi=g|˜yi=g)***

The disjunct (1) simply is false due to the construction of DC while disjunct (2) is absolutely uninteresting since it does not help to make any progress. Thus by guessing the problem itself has become harder due to reduced conditions on the guess-variable and it might become unsolvable therefore even if the original one does have a solution.

Guessing strategies for level one are comparably primitive. For convenience some variations are predefined.

basic_guess_sd (formula, sd, dclist, guess_var, taut_pred)
basic_guess (formula, dclist, guess_var, taut_pred)

The procedure *basic_guess_sd* is the strategy underlying all the other ones. It operates on a certain part of the formula body given by the structure-description sd. The name of the variable on which guessing will take place, a list of structure-descriptions how to select the domain-condition, and the tautology predicate have to be give as input. *basic_guess* does the same but operates on the whole body.

basic_guess1_sd (formula, sd, dclist, var_sd, taut_pred)
basic_guess1 (formula, dclist, var_sd, taut_pred)

Same as *basic_guess_sd* and *basic_guess* but with the guess-variable selected by a structure-description var_sd from the first domain-condition. In the end this allows guessing on terms too (we are not sure if this is a meaningful application of it).

std_guess_sd (formula, sd, dclist, guess_var)
std_guess (formula, dclist, guess_var)
std_guess1_sd (formula, sd, dclist, var_sd)
std_guess1 (formula, dclist, var_sd)

Similar to the above procedures with some standard predicate as tautology predicate.

1.4 Syntactical and logical reduction (Outlook)

A procedure *reduce* will eliminate superfluous parts of a formula. Its strategies include syntactical reduction (delete multiply occurring conjuncts/clauses) and logical reduction (delete conjuncts following from the other ones, eliminate false clauses etc.). For the latter kind knowledge about predicates and the theorem prover interface will be used. It will operate as a kind of rewrite transformation.

1.5 Transformation into recursive form

Besides guessing a transformation into recursive form is one of the most important LOPS strategy. It will take a recursion scheme of the form

[recursion-lemma, predicate-variable-list]

where recursion-lemma is a lemma of the form:

(domain-predicate -> (predicate-scheme <-> induction-scheme))

e.g.: ***(set(X) -> (p(X,Y)***
<-> (p(diff(X,Z),Y) & p1(Y,Z) & in(Z,X) & set(diff(X,Z)))))

and predicate-variable-list is a list of predicate variables occuring in the scheme (e.g. ***[[p],[p1]]***). It applies this scheme to a formula by instantiating the predicate-variable of the predicate-scheme by the head of the formula and creating a lemma which would transform the formula's body into the right hand side of the scheme. If the lemma is valid the strategy leads to a recursive program provided the open predicate-variables in the induction-scheme can be instantiated to something evaluable. The domain-predicate gives a domain classification of the output variables the recursion scheme is operating on.

Recursion schemes might be primitive recursion on integers or lists, set recursion, etc.. The amount of user control is not totally clear yet. It will be specified during implementation.

getrec_basis (problem, rec_scheme, sd)

1.6 Handling unevaluable predicates (Outlook)

For instance, the predicate ***x<yj*** where ***yj*** is an output variable is not evaluable. If, however, ***yj*** is bound to a fixed value in some other conjunct the whole expression becomes evaluable. A procedure *basic-ep* will replace ***yj*** by this value. Parameters are the description of the unevaluable predicate and of the conjunct where the binding occurs.

1.7 Reducing the number of output-variables (Outlook)

A procedure *basic-rnv* will try to express one output variable in terms of the others which might simplify the problem. Parameters are the output variable and a term expressing it.

Outlook: Level 2 and higher levels

Level 2 and higher levels are still open to development. They should include the following strategies (see ***[Bibel 1980]*** for theoretical background):

- A higher version of the GUESS-strategy
  It will use the I/O-Graph method (see ***[Neugebauer 1987]***) to select an appropriate output variable. Different tautology predicates found in the knowledge base may be tried and semantic knowledge on the predicates involved will be applied in order to find the domain specification. This requires a lot of knowledge to be stored in the knowledge base.

- Transformation into recursion scheme (GET-REC)
  Depending on knowledge about recursion schemes GET-REC will find an appropriate scheme to be applied to the formula.

- Find and replace unevaluable predicates (GET-EP)
  GET-EP shall use heuristic methods how to find and replace unevaluable predicates will be investigated in the future.

- Reduce number of output variables GET-RNV
  Try to express one output variable in terms of the others. Strategies how to find possible solutions from the preconditionsand the output condition shall be developed.

- Separate output condition w.r.t. to the output variables GET-SOC
  GET-SOC tries to split the output condition into parts where exactly one of the output variables occurs. The result would be a collection of single algorithms then which have to be combined in the end.

The TOP-level is intended to contain the LOPS-system as the user will see it. It is intended that the user will only call the command "LOPS" and everything else will be controlled by the system. A possible implementation for the implementation of the LOPS strategy is described in figure ***GN 1/87*** on the next page. Single strategies used within it (like GUESS, GET-REC etc.) will be implemented as TOP-level commands calling strategies from the highest implemented level. This allows to improve the system without changing it's general structure.

The TOP-level strategy of LOPS

Appendix: Some Demo Syntheses

We describe two syntheses as performed "by hand" using procedures as described above. Due to the fact that some of the procedures are still being refined and improved these syntheses might not run the same way when performed later. The examples are taken from ***[Fronhoefer 1985]*** and ***[Neugebauer 1986]***.

A.1 Maximum example

```
----------------------  PREDEFINED KNOWLEDGE ----------------------------

:define (kb_max ())

{  /* 1. store the definition as lemma and initialize synthesis    */

:newvar($max, (set(S) -> (  max(S,M)  <-> ( in(M,S) & le(S,M) )  )  ) ).
:store_lemma(max,$max).
:start_synth (max,$max).

  /* 2. Give definitions of the predicates occurring                    */
  /* :store_domain(set,set(L)).          */
  /* :store_ep_test(le,set,le(x,S)).     */
  /* etc.                                */

  /* 3. store a recursion scheme that often occurs with sets       */

:store_rec_scheme( set,[( set(X)              /* topic identification   */
                        -> (  p(X,Y)              /* the problem    */
                        <-> ( p(diff(X,Z),Y)      /* recursion      */
                           & in(Z,X)              /* identify Z     */
                           & p1(Z,Y)              /* the open rest  */
                           & set(diff(X,Z))       /* wellformedness */
                      )    ))
                     , [[p],[p1]]                 /* predicate vars */
                     ]).

  /* 4. store knowledge about the "le" predicate and set-recursion          */
  /*    the lemma is required by the getrec strategy it's truth is obvious  */

:store_lemma( m2, ( (set(S) & in(X,S) )
                -> (  ( in(Y,S)        & le(S,Y)      & ~(eq(Y,X)))
                  <-> ( in(Y,diff(S,X)) & le(diff(S,X),Y)
                     & less(X,Y)       & set(diff(S,X))   )
          )    )    ).
}.
```

```
:define (maxdemo())                          /* Version 11/87 */

{  :prints("THE MAXIMUM EXAMPLE\n-------------------------\nLoading...\n").
  :newvar($m0, :problem_of_synth(max)).
   :print($m0).
   :prints("\nSTEP 1: Guessing on M (the output variable)\n").
  :newvar($m1, :std_guess($m0,[[1]],M)).
   :print($m1).
   :prints("\nSTEP 2: Reduce in the success case\n").
  :newvar($m2, :t_reduce($m1,[[1,1]])).
   :print($m2).
   :prints("\nSTEP 3: Introduce Recursion in the failure case\n").
  :newvar($m3, :basic_getrec($m2,set,[2])).
   :prints("\n").
   :print($m3).
   :prints("\n-------------------------------------------------------------\n\n").
   :prints("DONE: the program is executable ").
}.


----------------------------  EXECUTION of maxdemo ------------------------------

THE MAXIMUM EXAMPLE
-------------------------

Loading...

(  set(S) ->  (  max(S,M)   <-> (  in(M,S)  &  le(S,M) ) ) )

STEP 1: Guessing on M (the output variable)

(  set(S)
->  (   max(S,M)
    <-> (  ( in(M,S)  &  le(S,M)  &  in(_M1,S)  &  eq(M,_M1) )
        | ( in(M,S)  &  le(S,M)  &  in(_M1,S)  & ~eq(M,_M1) )
)  )  )

STEP 2: Reduce in the success case

(  set(S)
->  (  max(S,M)
    <-> (  (  le(S,M)  &  in(_M1,S) &  eq(M,_M1) )
        | (  in(M,S)  &  le(S,M)   &  in(_M1,S)  & ~eq(M,_M1)  )
)  )  )

STEP 3: Introduce Recursion in the failure case

Checking if the following lemma is valid:

(  set(S)
->  (   ( in(M,S)   &  le(S,M)  &  in(_M1,S)  & ~eq(M,_M1)  )
    <-> ( in(_M1,S) &  set( diff(S,_M1))  &  in(M, diff(S,_M1))
        & le( diff(S,_M1),M)  &  p1(_M1,M) ) ) )
  Where the unknown predicates have to be instantiated

  Consulting Knowledge Base under the topics [set]
  no success with lemma max
  SUCCESS: Solution is lemma m2

( set(S)
->  (  max(S,M)
    <-> (  ( le(S,M)   &  in(_M1,S) &  eq(M,_M1) )
        | ( set(diff(S,_M1)) &  max(diff(S,_M1),M) & less(_M1,M) & in(_M1,S) )
)  )  )
-------------------------------------------------------------------------
DONE: the program is executable
```

A.2 Selection sort

```
/*------------------------  PREDEFINED KNOWLEDGE ----------------------------*/

:define ( kb_sort ())
{  /* 1. store the definition as lemma and initialize synthesis for 3 kinds */
   /*    of sorting programs: SELECTION SORT, INSERTION SORT, and QUICKSORT */

:newvar($sort,( (list(S) & list(L) )  -> ( sort(L,S) <-> ( perm(L,S) & ord(S) )   ) ) ).
:store_lemma (sort      ,$sort).
:start_synth (quicksort ,$sort).
:start_synth (selectsort,$sort).
:start_synth (insertsort,$sort).

   /* 2. Give definitions of the predicates occurring                    */

   /* :store_domain(list,list(L)). */
   /* :store_ep_test(isin,list,isin(x,L)). ????*/
   /* :store_e_func (first, list, first(S)). */
   /* :store_e_func (rest, list, rest(S)). */
   /* ---- etc.    */

:store_lemma(perm,( (list(S) & list (L) )
                  -> ( perm(L,S)
                  <-> (  ( empty(S) & empty(L) )
                     |  ( perm( diff(L,first(S)),rest(S)) & isin(first(S),L)
                       & list(diff(L,first(S)))  & list(rest(S))
          )   )    )) ).

:store_lemma(ord, (  list(S)
                   -> (  ord(S)
                   <-> (   empty(S)
                      | ( leq(first(S),S) & ord(rest(S)) & list(rest(S)) )
           )    )   )).

  /* 3. store a recursion scheme that often occurs with two lists   */

:store_rec_scheme(list3,[( (list(X) & list(Y))       /* topic identification */
                        -> (  p(X,Y)                  /* the problem    */
                        <-> ( p(diff(X,first(Y)),rest(Y))     /* recursion */
                           & eq(first(Y),Z)            /* identify Z     */
                           & p1(X,Z)                   /* the open rest  */
                           & list(diff(X,first(Y)))
                           & list(rest(Y))             /* wellformedness */
                         )  ))
                       , [[p],[p1]]                   /* predicate vars */
                       ]).

  /* 4. store knowledge about the "min" predicate (which is evaluable) and list-recursion  */
  /*    the lemma is required by the getrec strategy it's truth is obvious  */

:store_lemma(ord1,( (list(L) & ord(L) )  ->  ( eq(first(L),G) <-> leq(G,L) ) )).

:store_lemma(perm2,( (list(S) & list(L) & perm(L,S) )  ->  ( leq(X,L) <-> leq(X,S) )  ) ).

:store_lemma(min, (  list(S)  ->  ( min(S,M) <-> ( isin(M,S) & leq(M,S) ) ) ) ).
  /* :store_ep (min,list,min(L,x)) --result of a former synthesis see max */

:store_lemma( SUM_OF_perm2_min,
              ( ( list(L) & list(S)  & eq( first(S),X) )
                ->  ( ( perm(L,S) & ord(S) & isin(X,L) & leq(X,S)  )
                <->   ( perm( diff(L, first(S)), rest(S))  & ord ( rest(S))
                      & min(L,X)  & list( diff(L, first(S)))  & list( rest(S)) )
           ) )    ).
}.
```

```
/************************ SYNTHESIS OF SELECTION SORT ************************/

:define (selectsort())
{  :prints("THE SORTING PROBLEM: SELECTION SORT\n").
   :prints("-----------------------------------\n\n").
  :newvar($s0, :problem_of_synth(selectsort)).
   :print($s0).
   :prints("\nSTEP 1: Unfold definitions and normalize\n\n").
  :newvar($s1,:normalize(:unfold(ord,:unfold(perm,$s0)))).
   :print($s1).
   :prints("\nSTEP 2: remove contradictory and superfluous cases\n\n").
  :newvar($s2,:t_reduce($s1,[[3], [2],[1,3]])).
   :print($s2).
   :prints("STEP 3: case 1 is solved.\n   Perform guessing on the second one").
   :prints(" and fold definitions back.\n\n").
  :newvar($s3a,:std_guess1_sd($s2,[2], [ [2], [5] ], [1])).
  :newvar($s3,:fold_sd(perm,:fold_sd(ord,$s3a,[2]),[2])).
   :print($s3).
   :prints("STEP 4: since S is ordered the second case is contradictory.\n").
  :newvar($s4,:fold_sd(ord1,$s3,[2,2])).
   :print($s4).
   :prints("We therefore drop it and introduce recursion in the other one\n").
  :newvar($s5,:basic_getrec(:t_reduce($s4,[[2,2]]),list3,[2,1])).
   :print($s5).
 :prints("\n\n-----------------------------------------------------------------\n").
   :prints("DONE: we now have an executable recursive program").
}.


----------------------------  EXECUTION ---------------------------------------


THE SORTING PROBLEM: SELECTION SORT
-----------------------------------

( ( list(S) & list(L) )  ->  ( sort(L,S) <-> ( perm(L,S) & ord(S) ) ) )

STEP 1: Unfold definitions and normalize

( ( list(S) & list(L) )
->  ( sort(L,S)
    <-> ( ( empty(S) & empty(L) & empty(S) )
        | ( empty(S) & empty(L) & leq( first(S),S)
          & ord( rest(S)) & list( rest(S)) )
        | ( perm( diff(L, first(S)), rest(S)) & isin( first(S),L)
          & list( diff(L, first(S))) & list( rest(S)) & empty(S) )
        | ( perm( diff(L, first(S)), rest(S)) & isin( first(S),L)
          & list( diff(L, first(S))) & list( rest(S))
          & leq( first(S),S)  & ord( rest(S))  & list( rest(S)) )
)  )  )

STEP 2: remove contradictory and superfluous cases

( ( list(S) & list(L) )
->  ( sort(L,S)
    <-> (  ( empty(S) & empty(L) )
        |  ( perm( diff(L, first(S)), rest(S))  & isin( first(S),L)
           & list( diff(L, first(S)))  & list( rest(S))
           & leq( first(S),S) & ord( rest(S))  & list( rest(S)) )
)  )  )
```

STEP 3: case 1 is solved.
Perform guessing on the second one and fold definitions back.

```
( ( list(S) & list(L) )
->  ( sort(L,S)
   <-> ( ( empty(S) & empty(L) )
      | ( ( perm(L,S) & ord(S) & isin(_2,L) & leq(_2,S) & eq( first(S),_2) )
        | ( perm(L,S) & ord(S) & isin(_2,L) & leq(_2,S) &~eq( first(S),_2) )
)  )  ) )
```

STEP 4: since S is ordered the second case is contradictory.

```
( ( list(S) & list(L) )
->  ( sort(L,S)
   <-> ( ( empty(S) & empty(L) )
      | ( ( perm(L,S) & ord(S) & isin(_2,L) & leq(_2,S) & eq( first(S),_2) )
        | ( perm(L,S) & ord(S) & isin(_2,L)
                  & eq(first(S),_2) & ~eq(first(S),_2)
)  )  ) ) )
```

We therefore drop it and introduce recursion in the other one

Checking if the following lemma is valid:

```
( ( list(L)  & list(S) )
->  ( ( perm(L,S) & ord(S) & isin(_2,L) & leq(_2,S)  & eq( first(S),_2) )
   <-> ( perm( diff(L,first(S)), rest(S))  & ord( rest(S))
      & eq(first(S),_2) & p1(L,_2)
      & list( diff(L, first(S)))  & list( rest(S)) ) ) )
```

Where the unknown predicates have to be instantiated

Consulting Knowledge Base under the topics [list,list]
no success with lemma sort
no success with lemma perm
no success with lemma ord
no success with lemma ord1
no success with lemma perm2
no success with lemma min
SUCCESS: Solution is lemma SUM_OF_perm2_min

```
( (  list(S) & list(L) )
->  (  sort(L,S)
   <-> ( (  empty(S) &  empty(L) )
      | ( sort( diff(L,first(S)), rest(S)) & min(L,_2) & eq( first(S),_2)
        & list( diff(L,first(S)))  & list(rest(S)) )
)  )  ) )
```

DONE: we now have an executable recursive program

REFERENCES

[Bibel 1980]

Bibel,W.: *Syntax-directed, semantic supported Program synthesis* Artificial Intelligence 14 (1980), pp. 243-261.

[Bibel, Hörnig 1984]

Bibel,W., Hoernig,K.M.: *LOPS - a system based on a strategical approach to program synthesis.* in: Automatic program construction techniques , MacMillan New York 1984, pp. 69-89.

[Fronhoefer 1985]

Fronhoefer,B.: *The LOPS-Approach: Towards New Syntheses of Algorithms.* OEGAI-85, Vienna, Austria, September 1985, (H.Trost, J.Retti, eds.), Informatik-Fachberichte 106, Springer, Berlin 1985, pp. 164-172.

[Neugebauer 1986]

Neugebauer,G.: *Synthesis of sorting algorithms with the LOPS approach.*

[Neugebauer 1987]

Neugebauer,G.: *The I/O-Graph-Method: Program Synthesis for Sequentially Processed languages.*

[XPRTS]

Neugebauer,G.: *User's manual for XPRTS.*